ChatGPT

Audit Code Package Construction Guide

Part 1: SHEPARD.py – Context-Aware Control Script

The **SHEPARD.py** module is a lean supervisory script that orchestrates the SAT build process with minimal overhead. The Audit Supervisor should implement the following responsibilities in SHEPARD.py:

- **Build State Interpretation:** Dynamically read and interpret the current **SAT build state** to identify which module or phase is in progress. Use explicit context indicators (e.g. module IDs, phase flags) to determine the state of each component. This ensures the script always knows the active module and pending tasks, providing a real-time picture of progress.

- **Placeholder Propagation & Halting:** Enforce strict placeholder handling throughout the build. If a module produces or encounters a **placeholder** (an undefined or not-yet-computed value), propagate that placeholder forward rather than allowing undefined behavior. The script must **halt execution** at any point where a placeholder remains unresolved and is needed for the next step. No module should consume a placeholder as if it were final data – instead, SHEPARD.py should pause the workflow until the placeholder is replaced with actual data or a decision is made, preventing erroneous progression.

- **Root Hash Verification: Verify the SAT_ROOT_HASH** (a cryptographic signature or hash representing the canonical SAT foundational context) against the current version of the **Fundamental Intuitions of SAT**. SHEPARD.py should compute or retrieve a hash of the FUNDAMENTAL INTUITIONS reference content and confirm it matches the expected SAT_ROOT_HASH. If there is a mismatch (indicating the core foundational content has changed or the context is out of sync), the script must treat it as a critical blocker and prevent any further module progression. This guarantees that all processes are running against the correct, approved theoretical baseline.

- **Context Hash Consistency:** Prevent progression into any module whose context hash is outdated. Each module should carry a **context hash** representing the state of fundamental assumptions and upstream outputs when it was last validated. SHEPARD.py must compare this stored context hash to the current global context. If a discrepancy is found (meaning the module was built under older assumptions or data), **block execution** of that module and any dependent modules. The script should require a refresh or re-validation of the module under the updated context before continuing. This policy ensures no part of the SAT build runs on stale information.

- **Ambiguity Resolution via User Options:** Whenever the next step is **ambiguous** – for example, multiple possible branches, interpretations, or missing disambiguation – SHEPARD.py should refrain from auto-selection. Instead, gather the viable **options** and present them clearly to the user for a decision. In such a case, the script outputs the possible paths or interpretations, along with any relevant context for decision-making. SHEPARD.py then awaits explicit user input to choose an option. If no clear user direction is provided (or if ambiguity cannot be resolved), the script must halt

further progression. This guarantees that ambiguous situations are handled via deliberate choice rather than guesswork.

- **Status Communication (Module/Blocker/Option Tree):** Continuously communicate the build status in a structured, hierarchical format. SHEPARD.py should output a **tree view** of the current state, where each branch represents a module and sub-branches represent blockers or options:

- List each module with its status (e.g. completed, pending, or blocked).
- Under any blocked module, list the **blockers** (e.g. “Awaiting data X”, “Context mismatch”, “Placeholder Y unresolved”) as child nodes.

- If options are available for a blocker or ambiguity, list each **option** (decision branches or inputs needed) under that blocker.
  This clear tree structure allows the user and the Auditor to see exactly which part of the build is halted and why, and what choices (if any) are available to proceed. It should be updated in real-time as issues are resolved or as the state changes.

- **Minimal State Footprint:** Implement SHEPARD.py in a **stateless or lean-state manner**. Avoid any unnecessary global or persistent state that could cause bloat or cross-contamination between runs. The script should derive needed information from the source of truth (context files, module metadata, etc.) at runtime rather than storing long-lived mutable variables. Use local variables and straightforward logic so that memory usage is low and no hidden state persists beyond its necessity. This design prevents **memory dependency leakage**, ensuring that when a module finishes or resets, no residual data interferes with the next operation. In summary, SHEPARD.py must remain lightweight, with each run being as clean and deterministic as possible given the same inputs.

Part 2: PROTOCOLS_REF.py – Core Logic Protocol Definitions

The **PROTOCOLS_REF.py** file is a static reference defining all fundamental logic rules and protocols that govern the SAT build process. The Audit Supervisor should encode the following protocols in this file, structured for clarity and easy reference. These protocols will be invoked by SHEPARD.py and any auditing tools to ensure consistency and correctness:

- **Placeholder Handling Protocol:** Define standard procedures for dealing with placeholders in any module output or input. This should include:
- A uniform representation for placeholders (e.g. a specific object or token) recognized system-wide.
- Rules that any function or module must **propagate placeholders** forward if encountered (never transforming or dropping them silently).
- Conditions under which the presence of a placeholder triggers a mandatory process halt. For example, if a final output or a critical intermediate value is a placeholder, the protocol demands an immediate stop and flagging of that module as incomplete.

- Guidelines for resolving placeholders, such as acceptable sources of real data or user input to replace them. Until resolution, the protocols forbid treating that computation as finished.

- **Context Hash Policy:** Establish how context hashes are generated, compared, and used. This includes:

- Specification that the **context hash** is computed from the state of core SAT foundations (e.g. a digest of the FUNDAMENTAL INTUITIONS content and other global constants) and possibly the completion status of prerequisite modules.
- A requirement that every module's metadata stores the context hash under which it was last validated. If the global context hash changes (due to updates in foundational assumptions or major upstream changes), protocols state that all modules with a now-different stored hash are considered **invalidated**.

- Procedures for what to do when a context hash mismatch is detected: mark the module as needing re-validation or rebuild, and prevent its results from being used until it's updated. This ensures **context change handling** is systematic – no module is left running on outdated premises.

- **SAT Geometry for Ambiguity Resolution:** Provide guidance on leveraging the **SAT framework's structural geometry** to resolve ambiguities. In practice:

- Define what constitutes the "SAT geometry" in a computational sense (for example, the dependency graph or modular layout of the SAT theory implementation).
- Protocols should instruct that when multiple interpretations or execution paths are possible, the system should use the geometric relationships (module dependencies, theoretical constraints, ordering of phases) to eliminate inconsistent options. For instance, if one ambiguous branch would violate a dependency or create a circular reference in the module graph, the protocols would rule it out.

- If the ambiguity is not resolved by structural analysis alone, the protocol indicates the need for **user input**, aligning with SHEPARD.py's behavior. Essentially, PROTOCOLS_REF provides the logic to interpret the project's structure in order to guide disambiguation, ensuring that choices made are consistent with the overall SAT design.

- **Blocker Definition & Escalation:** Formally define what a **blocker** is and how blockers are to be handled:

- A *blocker* is any condition that prevents a module from progressing or being validated. Protocol examples: unresolved placeholders, failed logic checks, context mismatches, missing external data, or test failures.
- Categorize blockers by severity or scope. For example, **Critical blockers** (affecting foundational context or multiple modules) vs. **Local blockers** (affecting a single module's output).
- For each blocker category, outline the **escalation path**:
    - Minor/local blockers might simply pause that module and request a fix or data input.
    - Critical blockers (like a SAT_ROOT_HASH mismatch or fundamental inconsistency) should halt the entire build and require immediate attention.
    - Ambiguities are a special kind of blocker handled by presenting options to the user.

- Define a **ranking or priority system** if multiple blockers occur: e.g. address higher-ranking blockers (foundational issues) before lower ones. The protocol ensures the Audit Supervisor and Auditor know which problems to tackle first. It also helps SHEPARD.py present the blocker tree in a logical order (highest priority at top).

- **Symbolic Logic Consistency Checks:** Establish protocols for verifying logical consistency across modules:

- Identify key logical assertions that must hold true in the SAT build. For example, if module O2's output logically requires module O1 to have run (A implies B), or if two modules produce results that must not conflict, these relationships should be codified.
- In PROTOCOLS_REF, include checks or formulas representing these constraints (potentially pseudo-code or descriptions that the Auditor can translate into actual test code). The protocol might say: *"For any modules X and Y with defined logical dependency, verify that X is complete and consistent before Y runs; if Y runs first or contradicts X, raise a logic error."*

- The **symbolic logic checks** ensure there are no internal contradictions. If a contradiction is found, it is treated as a blocker (enforced via the blocker protocol). By encoding these rules, the system guards the epistemological soundness of the build – no progression if the theory's logical structure is violated.

- **Mathematical Sanity & Module Validation Criteria:** Outline the quantitative sanity checks every module must pass to be considered valid:

- Define general **sanity conditions** (e.g. no output producing infinities or NaNs unless expected, values staying within physical bounds, conservation laws not broken without justification, etc.). These conditions can be generic across modules or specific to certain module types (for instance, a module computing a probability must output a number between 0 and 1).
- For each module (or category of modules), list **validation criteria**. This may include checking against known results or thresholds. Example: *"Module O9's output spectrum must not be negative-mass values; verify all masses > 0. If any fail, invalidate the module."* or *"Geometric consistency check: if module O10 computed a particle mapping, ensure all referenced IDs exist in data."*

- Specify that module validation requires passing all such checks. If any check fails, the protocol demands that the module's status be **marked invalid** and a blocker is raised for that module. This way, only mathematically and physically sensible results are accepted into the build.

- **Code and Data Consistency Guidance:** Provide guidelines to maintain consistency between code, data, and documentation:

- Ensure **nomenclature and references** remain consistent. For instance, if the theory documentation or FUNDAMENTAL INTUITIONS defines certain terms or constants, the code and data should use the same definitions (no divergent naming or values). The protocol might instruct the Audit Supervisor to cross-verify that variables in code align with the theoretical terms in documents.
- Instruct that any change in core logic in code should be reflected in the PROTOCOLS_REF documentation comments and, if applicable, in the FUNDAMENTAL INTUITIONS or associated documentation. This keeps the entire project philosophically and technically synchronized.
- Define a process to check **data integrity**: if external data files (e.g., lookup tables, experimental inputs) are used, they must be the correct version for the code. The protocol could, for example, require including a checksum or version tag in data files that the code verifies before use.
- Emphasize internal consistency: the code within and between modules should not duplicate contradictory logic. PROTOCOLS_REF can note that if two modules handle overlapping domains, they

must do so in a coordinated way defined here. This guidance ensures no forked logic or accidental inconsistencies creep in during development.

Packaging and Verification Process

After implementing **SHEPARD.py** and **PROTOCOLS_REF.py** according to the above instructions, the Audit Supervisor must package them for audit and ensure an iterative verification loop as follows:

1. **Package Assembly:** Bundle the two files, **SHEPARD.py** and **PROTOCOLS_REF.py**, into a single **Audit Instruction Package**. Ensure the package is complete (both files present and up-to-date) and ready for review. Include any brief README or manifest if needed to clarify versions or dependencies, but avoid extraneous content. The goal is a self-contained package that an independent Auditor can examine.

2. **Submission to Auditor:** Provide the Audit Instruction Package to the designated **Auditor** for assessment. Clearly communicate that this package is to be reviewed for compliance with all requirements. The Auditor should be made aware that SHEPARD.py is the active control script and PROTOCOLS_REF.py the reference of rules, and that both are to be read in conjunction.

3. **Auditor Verification Checklist:** Instruct the Auditor to rigorously evaluate the package on several fronts:

4. **Correctness:** The Auditor checks that the code in SHEPARD.py correctly implements each responsibility outlined (e.g. does it actually halt on placeholders? Does it check the SAT_ROOT_HASH correctly? etc.), and that PROTOCOLS_REF.py accurately defines the required protocols. They may run tests or simulations of scenarios to verify halts and outputs occur as specified.
5. **Internal Consistency:** The Auditor ensures that SHEPARD.py's behavior aligns with PROTOCOLS_REF protocols. Any mismatch (for example, if PROTOCOLS_REF says to handle a case one way but SHEPARD implements it differently) must be flagged. The two parts should be fully consistent with each other.
6. **Code Sufficiency:** The Auditor confirms that the code covers all scenarios and requirements without omissions. Every bullet point in these instructions should be translated into code or structured logic. There should be no gaps (e.g., if a type of blocker can occur, the code must have a branch to handle it). Also, the code should be efficient and minimal as prescribed.

7. **Philosophical Adherence:** The Auditor verifies that the approach and rules embodied in the code remain true to the **SAT foundations** and philosophy. This means checking that nothing in SHEPARD.py or PROTOCOLS_REF.py contradicts the fundamental principles of the SAT framework. For example, the Auditor would confirm that any decision-making or validation criteria are in line with SAT's core theoretical intentions (as outlined in the Fundamental Intuitions document [1]). Any sign that the code permits violations of SAT's basic tenets or pillars would warrant a correction.

8. **Feedback and Iteration:** If the Auditor finds any issue – whether a logic error, an inconsistency, insufficient handling of a case, or a philosophical discrepancy – they must document these findings and return the package to the Audit Supervisor with clear notes on what needs improvement. The Audit Supervisor is then responsible for addressing each point:

9. Fix code bugs or implement missing features in SHEPARD.py.
10. Update PROTOCOLS_REF.py to clarify or correct protocol definitions.

11. Improve alignment with SAT theory if the Auditor noted a philosophical issue (this may involve consulting the SAT Fundamental Intuitions and adjusting the logic to better reflect those principles). Every change should be made in a controlled manner, ensuring other parts of the package remain consistent after the fix.

12. **Repeat Until Convergence:** The revised package is resubmitted to the Auditor for another round of checks. This **Audit loop** (package -> audit -> feedback -> refine) continues iteratively until the Auditor finds no remaining issues and **the end user confirms that the package has converged on a satisfactory state**. “Convergence” here means the code and protocols are fully correct, consistent, sufficient, and in harmony with SAT’s foundation, with no ambiguity or blocker unaddressed. Only once the user (or project owner) explicitly confirms that all requirements are met and no further changes are needed will the process conclude.

By following these instructions closely, the Audit Supervisor will create a robust two-part code package. This package, once vetted, ensures the SAT build system operates with clarity, rigor, and fidelity to its theoretical foundations. The iterative audit cycle guarantees that the final deliverables are of the highest quality and epistemologically sound before final user acceptance. [2] [1]

---

[1] [2] FUNDAMENTAL INTUITIONS.txt
file://file-NChkb4dhRtY66doviwRbqU